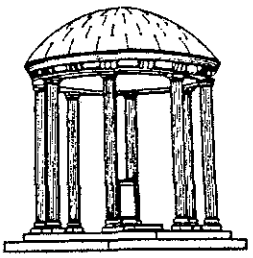
FPC: A Translator for FP

TR88-027

May 1988

Edoardo S. Biagioni



The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175

FPC: A Translator for FP

by
Edoardo S. Biagioni

Abstract

FP is the applicative programming language introduced by John Backus in 1978. FPC is a translator that accepts programs written in FP and translates them to C programs which can be compiled using an ordinary C compiler. This report includes the FPC user manual, the reference manual for the FP language accepted by the translator, and the installation and maintenance guide for FPC.

Table of contents

1 Introduction

Ten years have passed since the FP programming language was first described by John Backus [Ba 78]. FP is a strict functional language with no variables or named parameters. Functional languages in general lack assignments and side effects; strict languages evaluate the arguments to a function before they are passed (lazy languages in contrast evaluate arguments when they are actually used). The absence of named parameters is unique to FP. FP puts very few restrictions on evaluation order, and in fact encourages the use of constructs that would be executed in parallel on parallel machines. FP is also sufficiently abstract that several different implementations are possible for any given construct; as a result, an FP program can be efficiently implemented on very different architectures.

Various FP interpreters have become available in the time since 1978; in particular, the Berkeley interpreter [Ba 87] runs under the Unix BSD operating systems and is available to every Unix BSD site. This interpreter and others like it have several disadvantages, chief among which are speed and usefulness for actual programming.

Speed is a common concern in programming language implementations. As shown by the IFP project [Ro 87], a good interpreter can be much faster than the Berkeley FP system. Even a straightforward compiler, on the other hand, runs faster than a highly optimized interpreter such as IFP; an optimizing compiler could run orders of magnitude faster.

Interpreted programs have to be executed within the interpreter itself, so stand-alone programs are unrealistic. Most real-world, useful programs need to be stand-alone, that is, to be usable outside the environment in which they were developed. The smalltalk programming environment [GR 83] is no exception: in the author's experience, it is used mostly for writing prototypes of applications that are later ported to other languages.

An obvious solution to both of the above problems is a compiler for FP. However, any compiler that produces native code is by definition not portable among different systems. A portable FP compiler needs to compile to some portable intermediate language which can then be compiled locally on the desired system. This is the approach taken by FPC, which has C as the intermediate language. The advantages of C are the wide availability of compilers and the portability of programs written in C. Since FPC rewrites an FP program into an equivalent C program we call it a translator instead of a compiler.

The remainder of this report consists of three main parts. The first part (Sections 2 - 5) is the FPC user manual, which shows how to write FP programs and invoke FPC. This part also gives suggestions for debugging FP programs.

The second part (Sections 6 and 7) is a reference manual for the FP programming language accepted by FPC. This language has been made as close as possible to the

description in [Ba 78], but the several differences are noted and discussed.

Sections 8 and 9 are the installation and maintenance guide for FPC. They are not needed by people who just program in FP, but may be of interest to the dedicated FP programmer who needs to know or modify the algorithms used by the translator. Maintainers would also be well-advised to read appendix D.

FP programmers and prospective FP programmers might be interested in Section 10, which records my own view of the current state of FP and the direction it's moving in. It is my hope that this will motivate people to use FPC and contribute to progress in the field of functional programming.

The appendices are referred to in the text and should be consulted as necessary.

2 Getting started

This is a tutorial introduction to FP using FPC. If you are already familiar with FP, you may wish to skip this section and the next, or just skim them to make sure you can read the syntax. If you are not familiar with FP, this section should help you get to know the language so you can at least read simple programs. Section 3 should help you learn to write simple programs -- only experience will teach you how to write complex programs!

Let's start with the factorial function as defined in Backus's Turing award lecture. He defined it approximately as follows (the slight difference is due to the different syntax) :

```
Def sub1 - o [id, _1]
Def eq0 eq o [id, _0]
Def fac eq0 -> _1; * o [id, fac o sub1]
```

If the above were the contents of the file fac.fp, you could compile it and run it as follows:

```
% fpc -m fac.fp
% cc -o fac fac.c -lfp
% fac
4
24
%
```

where the prompts and the result are written differently to show that they are printed out by the computer; everything else you had to enter by hand.

So we have a function that computes the factorial (run it over inputs other than 4, to make sure). Let's go over it step by step.

The factorial function is defined to return 1 for an inputs of 0, and the product of its input and the factorial of one less than the input for larger numbers. In other words,

factorial (0) = 1
factorial (n, n > 0) = n * factorial (n - 1)

What all this means is that factorial returns the product of all the numbers up to and including the number it was given as an input.

```
Def sub1 - o [id, _1]
```

"Def" means we are defining a function, the function **sub1** which subtracts one from its argument. The function needs to be read from right to left ("backwards"). The first step (the first expression to be reduced) is "[id, _1]" -- remember that we read from right to left! When this constructor is applied to any value, it returns <original value, 1>, since: "id" represents the argument to an expression; "_1" means the number 1 ("_", underscore, tells us that we have a constant expression); and a constructor "[f1, f2]" means return the two-element vector (pair) obtained by applying each of f1 and f2 to the argument. In other words, if ':' means application, we have

```
[id, _1]: value                                    rewrites as
< id: value, _1: value>                            rewrites as
<value, 1>                                         (final value)
```

Value is whatever value was input to the expression. For instance, [id, _1]: 4 rewrites to <4, 1>.

We continue our right-to-left reading of sub1. The pair (two-element vector) returned by the constructor is then given to the operator '-', which is an expression that always returns the difference between the elements of a two element-vector. The 'o' you see between the minus and the constructor is the composition operator: it says "apply the left-hand expression the result of applying the right-hand expression to whatever argument is supplied". To put it another way, '(a o b): x' in FP is the same as a (b (x)) in other programming languages.

In short, sub1 takes a number, pairs it with a 1, and takes the difference. Which is the same as saying that sub1 returns one less than its input. Here is a concrete example:

```
sub1: 17                                           rewrites as
(- o [id, _1]): 17                                 rewrites as
- : ([id, _1]: 17)                                 rewrites as
- : (<id: 17, _1: 17>)                             rewrites as
```

```
- : (<17, 1>)                                    rewrites as
16                                               (final value)
```

Now that you know how sub1 works, we can discuss eq0:

```
Def eq0 eq o [id, _0]
```

eq0 is structurally very similar to sub1: it compares its argument to 0 and returns whether equality holds or not. Unlike sub1, the value returned is a boolean, i.e., one of T or F; sub1 returns a number.

```
Def fac eq0 -> _1; * o [id, fac o sub1]
```

fac shows us a new functional form (we have already seen composition, construction, and constant): conditional. A conditional has the form *predicate -> then-part ; else-part*. For fac, *predicate* is eq0; *then-part* is the constant 1; and *else-part* is * (times) composed with the constructor [id, fac o sub1]. If *predicate* applied to the argument evaluates to true, we return the result of applying *then-part* to the argument; otherwise we return the result of applying *else-part* to the argument. In the case of fac, if 'eq0: value' returns true (which only happens if the value supplied is 0), we return '_1: value', which returns 1. Otherwise we return the product of the value supplied and the factorial of one less than the value.

The above is a recursive implementation of the factorial function that resembles the function's inductive mathematical definition. We could re-write fac as follows:

```
Def fac /* o iota
```

The slash means insert, i.e. insert the following operation (times, in this case) between every pair of elements in the vector that is being given as value. For instance, '/+: <1, 2, 3, 4>' gives 1 + 2 + 3 + 4, i.e. 10; similarly, '/* <3, 17, 22>' gives 3 * 17 * 22 or 1122. Iota is a primitive function which takes a positive integer and returns a vector of all the numbers from one to its argument, e.g. iota: 6 returns <1, 2, 3, 4, 5, 6>. It is left as a trivial exercise for the reader to check that this new factorial function also works (note that '/f: <el>', i.e. insert applied to a vector of length 1, returns el) and produces the same results as the other definition for all n > 0.

In the above we have come across two distinct terms: function and functional form. A function is a normal function; it takes a single argument and returns a single result. The argument, the result, or both, may be structured: the argument to '-' and eq must be a pair, the result of iota is a vector.

A functional form, on the other hand, takes both an argument and one or more functional expressions (hereafter called expressions). For instance /, the insert functional form, takes '*' as the expression, then takes an argument and inserts its

expression between successive elements of the argument. Similarly the **constructor** takes any number of expressions (e.g., 'id', 'fac o sub1', etc.) and an argument and applies each expression to the argument, building a vector of the results. The **constant** functional form takes a value for an expression and, when given an argument, ignores it and returns the expression instead. **Conditional** takes three expression, applies the first one to the argument, then depending on whether that evaluation returned true or false, returns the result of applying either the second or the third expression on the argument. Finally, **compose** takes a left and a right expression and composes them, returning the value produced by the left-hand expression when given the result of applying the right-hand expression to the argument. Section 6 contains more detailed descriptions of all the functional forms and examples for each of them, should you be confused.

An important functional form that we have not yet seen is aa (apply to all, written as α, the greek letter alpha, in [Ba 78]). The expression 'aa f: $<x_1, \ldots x_n>$ returns a vector of length n, where each element of the result is the result of applying f to the corresponding x_i. For example, the function senior determines, given a vector of ages, how many of those ages are 65 or over:

```
Def senior /+ o aa (>= o [id, _65] -> _1; _0)
```

The vector returned by the first expression (the one in parentheses, on the right) has a 1 for each element of the original vector that was ≥ 65, and 0 for all others. We then sum all the values to get the desired result. E.g.

```
senior: <47 92 21 48 65 64>                    rewrites to
/+: <0 1 0 0 1 0>                              which rewrites to
2
```

You now know enough to read simple FP programs. If you want to practice some more, Appendix A has some sample programs. Start with the easy ones and refer to the comments if you have trouble understanding anything. The next section should get you started writing (as opposed to just understanding) FP programs.

3 Introduction to FP

Once you are through this section, you should not only be itching to program in FP, but hopefully also have enough knowledge to do so. For this section, I will assume that you are seated at a terminal where you can try things out. If you aren't, you will need to experiment later on.

We have already seen a sample terminal session using fpc. We reproduce it here:

```
% fpc -m fac.fp
% cc -o fac fac.c -lfp
% fac
4
24
%
```

The first command (fpc -m) translated the file fac.fp to the file fac.c specifying (-m) that the function with the same name as the file (i.e., the function fac) was the one to be applied to the argument entered by the user. The second command invoked the C compiler cc on the file fac.c producing as output the executable file fac. Notice we had to link in the FP runtime system by using the C compiler switch -lfp. The runtime system is where all the primitive functions, such as iota or eq (seen in Section 2), are defined. On some systems the runtime system may not be installed, and the second line would then look as follows:

```
% cc -o fac fac.c fp.o
```

where fp.o is a standard file distributed with FPC. The only difference here is that you have to explicitly specify the runtime system, which is file fp.o. You only need this form if the earlier command did not work.

The third line invokes fac. The input could have been any number between 0 and 12 -- the factorial of 13 cannot be represented using 31 bits, so the computation overflows (try it).

The result is printed to the standard output; in this case, to your terminal.

As a first exercise, you should try compiling and running the alternative version of fac shown on page 5. You can put it in fac1.fp, if you wish to avoid disturbing the file fac.fp, but you must then remember to name the function fac1, since FPC tries to call the function with the same name as the file it is in. Play around a bit, try, for instance, omitting the -m switch or the run-time library. You will get various error messages and, next time you forget something, you will be able to tell from the error message what the problem is.

For an easy exercise, try writing a program to compute the fibonacci function, which is defined as follows:

fib (0) = 0; fib (1) = 1;
fib (n, n > 1) = fib (n - 1) + fib (n - 2)

The program should closely mirror the inductive definition of fib.

We now return to the "senior" program introduced in Section 2. Suppose we wanted

to modify the program so it would take as argument not just a vector of ages, but a particular (given) age and vector of ages (combined in a pair). This function should return how many ages are greater than the given one. The numbers need not be ages, of course -- they could be salaries, or scores on a test, or anything else. Here is a first definition for numgreatereq:

```
Def numgreatereq /+ o aa (>= o reverse -> _1; _0) o distl
```

The function distl takes a pair of which the second element must be a vector and distributes the first element of the pair to each element of the second. In the reduction of the above program the first step would be, for instance

```
distl: <43, <47, 11, 45, 41>>                    which reduces to
<<43, 47>, <43, 11>, <43, 45>, <43, 41>>
```

The business with "reverse" is somewhat messy. After we have ditributed the first element, we need to reverse the pairs so we can use >=, as we used in the program 'senior'. So we apply reverse to every one of the pairs we produced. An alternative is to switch the position of the two elements in the original argument, and use distr instead of distl:

```
distr: <<47, 11, 45, 41>, 43>                    rewrites to
<<47, 43>, <11, 43>, <45, 43>, <41, 43>>
```

This can be achieved simply by reversing the input! So the final function definition is

```
Def numgreatereq /+ o aa (>= -> _1; _0) o distr o reverse
```

What we just went through is an example of program transformation, where a program is rewritten but performs exactly the same function. In this case, we moved reverse from inside the apply-to-all expression and put it to the right of distl, which we changed to distr. The result is exactly the same program, but somehow clearer and probably more efficient. If you are really interested in program transformations, you should read [Ba 78], since it has many more interesting and thorough examples.

The example did show us the new primitives distl, distr, and reverse. Reverse, by the way, works on vectors of any length and on the empty vector, not just on pairs. Distl and distr always take pairs consisting of a vector and an arbitrary object, but arranged in the opposite orders: for distl the object is to the left, for distr the object is to the right.

One of the nice things about FP is the symmetry of all its primitive functions that deal with vectors. For instance, distl corresponds exactly with distr; we also have the vector extension functions apndl and apndr, which work as follows:

```
apndl: <ext, <x1, x2, x3>> returns <ext, x1, x2, x3>,and
```

apndr: <<x1, x2, x3>, ext> returns <x1, x2, x3, ext>.

In addition, we have left and right selectors, as in:

2 : <x1, x2, x3, x4, x5, x6> returns x2, and
2r: <x1, x2, x3, x4, x5, x6> returns x5.

and left and right tail:

tl : <x1, x2, x3, x4> returns <x2, x3, x4>, and
tlr: <x1, x2, x3, x4> returns <x1 x2 x3>.

You see that, unlike LISP's lists, FP vectors are treated as symmetric structures: anything that can happen on the left can happen on the right, and vice versa.

The next exercise is to write a function that will concatenate two vectors, i.e., such that concat: <<a, b, c>, <d, e, f>> will return <a, b, c, d, e, f>. This is the same as done by the built-in function append, by the way. To implement concat we will use append, (that would be cheating), but rather apndl and apndr. The crucial observation is that if we apply either one of the functions to the argument for concat, we get an asymmetric result, i.e. <<a, b, c>, d, e, f> for apndl and <a, b, c, <d, e, f>> for apndr. But if we take the second form and use /apndl, we can obtain the desired result, since /apndl <x, y, <z>> gives <x, (apndl <y, <z>>)>, which works out to <x, y, z>, as desired.As you can see from this example, insert is right associative.

But can such a symmetric language have only a right-associative insert operator? Well, you guessed it. We not only have the / form of insert (which we call left insert since results keep moving leftward), we also have a right insert, which we write \ So our concat program can be written in either of two ways:

Def concat /apndl o apndr, or
Def concat \apndr o apndl, which is equivalent.

In fact, we even have a third way of writing insert: \/. This insert is also known as tree, or tree insert. The trick in **tree insert** is that the pairing of elements (and results) is not known to the programmer and may be indeterminate. Tree insert should be used at all times when the associativity of the operator doesn't matter (as in, for instance \/+ or \/*), since that allows the implementation to pick the best possible strategy for implementation; for instance, in a parallel machine \/f might reduce much faster than either \f or /f, since more applications of f might be reduced in parallel.

We can observe the effect of the different inserts by inserting the function id (which just returns its argument) with any old vector as argument:

/id : <a, b, c, d, e, f> returns <a, <b, <c, <d, <e, f>>>>>,

```
\/id: <a, b, c, d, e, f> returns <<<a, b>, <c, d>>, <e, f>>,
\id : <a, b, c, d, e, f> returns <<<<<a, b>, c>, d>, e>, f>.
```

These results may not seem obvious at first, but they should become clear after some reflection. Try experimenting with different data and operators other than id, e.g. +, *, apndl, apndr, append, distl, distr, 2 (selector 2), 2r, and so on.

We have two more very convenient functinal forms: bu and bur. bu stands for binary to unary, and bur for binary to unary on the right. They are used to simplify operations such as adding a given value to the input, comparing the input to a given value, and so on. Recall that eq0 was written as "eq o [id, _0]"; it is easier to write it as "(bur eq 0)", which is equivalent. Both bu and bur take a constant value and an expression, build a pair which contains both the constant value and the input, then apply the expression to the pair. That is, "(bu x ob)" is the same as "x o [_ob, id]", while "(bur x ob)" is the same as "x o [id, _ob]". For commutative expressions such as eq and +, bu and bur are equivalent; for noncommutative expressions such as -, apndl, and distr, only one of the two forms is useful. bu and bur are typically used in functions such as sub1, which becomes "(bur - 1)", as opposed to "- o [id, _1]". bu and bur are just notational conveniences, but they are used quite often. Here is a new version for senior:

```
Def senior /+ o aa ((bur >= 65) -> _1; _0)
```

The above completes the discussion of the functional forms. Once again they are: apply-to-all (aa), composition (o), conditional (-> ;), constant (_), constructor ([ ]), insert (/, \, \/), and while (while), which is discussed in section 6.To gain practice in programming FP, I recommend that you try and program all of the following, sample solutions for which are given in Appendix A:

An efficient fib, where you start computing fib(0) and fib(1) then pass the results on to a recursive invocation of yourself; the previous, inefficient version of fib started by requesting the values of fib(n - 1) and fib (n - 2).

The function flatten, defined such that flatten: <<a, b>, c, <>, <<<d, e>>, f, <>>> returns <a, b, c, d, e, f>. You will probably find the function append (which is like concat except it merges any number of elements together, not just two) and the functional form aa useful. Only use recursion where necessary.

A function innerproduct, which given a pair of vectors of the same length computes the sum of the products of the elements of the vectors. You might find it useful to employ the function trans (transpose), which given an n-sized vector with m-sized vectors as elements returns an m-sized vector with n-sized vectors as elements, as in trans: <<a, b, c, d>, <1, 2, 3, 4>, <w, x, y, z>> returns <<a, 1, w>, <b, 2, x>, <c, 3, y>, <d, 4, z>>. Again, do not use recursion.

4 Using FPC

This section is the reference manual for fpc. It tells you all about how to use the translator and what you can do with it. To find out more about how FPC works, refer to sections 8 and 9 and to Appendix D.

Fpc is meant to let you write substantial programs in FP. As such, there are lots of options to let you say exactly what you want, and facilities for breaking a program into several independent files.

You already know that fpc is a translator: it takes FP source files of the form filename.fp and produces equivalent C source files of the form filename.c. Equivalent means that the C files, when compiled and linked with the appropriate run-time system, give a program that behaves in the same way as your FP source program.That is, they accept an input, apply the 'main' function to it, and print the result. The only reason FPC is different from a true compiler is that you have to manually compile the C files using cc (or any other C compiler). It also means that it's a really bad idea to use names such as "int" or "if" for your function names, since the C compiler will complain that you are using keywords as function names.

One advantage of translating to C is that we get to split our FP programs into independent files, since C lets you divide programs into separate modules. This lets you have libraries, for instance, that can be compiled independently and later linked into your program. One such library is the run-time system, normally found in a file named fp.o. This file is usually installed as the fp library on your system, where it can be loaded using the option -lfp. If it is not installed, you have to load it explicitly by naming it in the 'cc' command.

In what follows, remember that each switch only affects translation of the next file name to follow it on the command line. In other words, "fpc -n x.fp -n y.fp" is different from "fpc -n x.fp y.fp", because in the second case only x.fp is affected by the -n switch.

Since the modules can be compiled independently, FPC needs to know whether or not to generate code for a C procedure called "main" that will do all necessary input and output and will call your program. This procedure may only be defined once in any given C program, so FPC needs to know whether you want your file compiled as a library (subsidiary) module, or whether you want it to be a main module which defines the function that will be executed at top level when you run the program. The switch -m indicates that the file should be a main module; its absence means the file should be a subsidiary module. By default, the top-level function is the one named as the file it's in (without the .fp extension), but if you wish to tell FPC that the top level function in module file.fp is **fun**, just say so:

```
% fpc -mfun file.fp
```

will do the trick. The more usual call,

```
% fpc -m file.fp
```

tells FPC that the top level function is a function named **file**. If none of your functions are called file, the loader will complain that it cannot find the function. As a further example, we translate files a.fp and b.fp, where a.fp has the top level function, which is named **toplev**:

```
% fpc -mtoplev a.fp b.fp
```

To get more feedback while translating, the -v switch tells you which version of FPC you are using, then echoes the names of the functions being translated.

The -s switch generates code to give you some runtime statistics about the amount of storage used. It is incompatible with the -n or -lnfp switches (described later), and can only be used in conjunction with the -m switch.

FPC is meant to be useful for programs that work on arbitrary text, not just FP expressions. Admittedly, simple programs work on plain numbers or vectors of numbers, but more useful programs have to read and possibly parse their own input directly. To support this, FPC provides several options that are only valid if the -m switch is also used. Switches -i and -o tell FPC that the FP program should read the input directly (without trying to parse it as an FP object) or that it will produce as output a string which should be printed directly (without surrounding quotes), respectively: -i stands for string Input, -o stands for string Output.A string is a vector of character values; see also section 6. In fact, if the -o option is used, the output of the program need not be a string; it can be a vector of pairs <<filename1, string1>, <filename2, string2> ... <filenameN, stringN>>. This kind of output means that instead of outputting a single string to the standard output (usually the screen), the program can direct that the various strings be written to the named files. Notice that writing destroys any pre-existing files with the same name. No two file names may be the same; an empty vector (nil) in the position of a file name will print the corresponding string to the standard output.

Switch -a is even more complicated. If you've read the Backus Turing Award lecture [Ba 78], you may have a vague idea of what an AST (Applicative State Transition) system is. Well, -a tells FPC that your main function is the function to be used in an arrangement similar to an AST system. The function gets called with the pair <input, state>, and must return a pair <output, new-state>. The function gets called over and over again until it returns a new-state of nil, the empty vector. On each cycle input is read from the standard input (usually the user's terminal), the function is applied to the pair, the output part of the result is written to the standard output (usually the user's terminal), and the state part of the result is passed on to the new invocation of the function. The first time it is called, the state is nil and the input is nil; for all other

invocations the state is whatever was returned by the previous call, and the input is whatever was read.

If -a is combined with -i, the input will always be a string of exactly one character, the next character found on the standard input. If -a is combined with -o, the function may write to files by providing pairs of file names or strings, as described above, or do direct output of strings.

The -p switch can also only be used in conjunction with -m. If it is used, the program checks whether any command line arguments or options are present; if they are, the program is run immediately using an input value of <>, without waiting for data to be given on the standard input. You can use the primitive function *arguments*, described in section 8, to find out what the arguments or options are. If neither options nor arguments are used in the call to the program, the program proceeds normally by reading the standard input.

There are several switches for setting the level of debugging and error checking. They are -d, -e, -t**function**, -n. These may all be used independently of any -m switch. Normally (if none of these switches are given) FPC checks for such things as bad argument type in the invocation of a primitive (i.e., passing a pair of symbols to *, division by 0, passing a non-vector to length) and keeps a stack of function invocations so that, should an error be detected, you can find out where the error occurred and why it happened. This is covered in more detail in the next section.

This error checking unfortunately takes time.To speed up production programs, we provide the option -n.When translating with this option, the compiler does not produce code to check that the argument to an apply-to-all or an insert is a vector, nor does it keep a stack of function invocations. If a type mismatch occurs, your first indication might be a system error; on Unix, this may be a segmentation fault or a bus error. You might also get the wrong results, with no notice given. In general, it is a good idea to thoroughly debug your programs with the normal settings before producing a release version with the -n switch. If you do have a problem, you can translate the program again without the switch and run it on the same input -- the error should manifest itself in an intellegible fashion. On the other hand, programs translated with the -n switch average about 30% faster and 30% smaller than programs translated normally. A program translated with -n will normally be linked to nocheckfp.o instead of the standard fp.o; this can be achieved by using the -lnfp switch instead of -lfp.

If you are still debugging a program, you want more, as opposed to less, information when an error occurs, and sometimes even when it doesn't. Translating with the -t switch (which has to be used in the form -t**functionname**) tells FPC that the function **functionname** must be traced; that is, it should print out when it is being called and when it is returns, the data that it is given and the result that it returns.

The debug switch, -d, is essentially the same as -t, except it says that every function

(not just one or a few) should be traced. Usually -d is impractical for large modules, but very useful for small and medium-sized modules. Programs which manipulate large data items can use -e (entry/exit) instead of -d. With -e, functions announce their start and end but do not print out ther inputs or results. Combining -e with -t can be very productive, for large programs.

If you are determined to see every step of the way, you can link your program to debugfp.o (instead of fp.o or the library -lfp), which may be accessible through -ldfp. debugfp.o shows you the entry and exit of every single call to a primitive, and shows you exactly what happens. Notice that even debugfp.o does not show you every invocation of a functional form -- functional forms are compiled in-line, and are not displayed under any debugging mode. For instance, if you were debugging the expression "aa +" applied to the data <<1, 2> <3, 4>>, you would see something like

```
entering plus, object is <1, 2>
exiting plus, result is 3
entering plus, object is <3, 4>
exiting plus, result is 7
```

In the above you can see that apply-to-all works front to back along the vector. This is very implementation dependent and may change within a given implementation, so you would be wise to ignore the order in which things happen. The positive side of this is that FP programs do not have side effects, so the result of an expression is the same no matter what order the subexpressions are executed in.

5 Debugging

In section 4 we described the -d, -t and -e switches and how they can be used for debugging. This section will address the issue of debugging in greater detail.

FPC provides support for debugging FP programs as well as for debugging itself. The former include the call stack and the debug/trace modes mentioned above; the latter include the reference count summary and the dfp library, and are described in section 8.

A bug in an FP program will generally be discovered when an incorrect output or an error condition is produced for a reasonable input. The incorrect output is then due to one or more bugs in specification, design, or coding, as in normal programming. Bugs in specification or design are not substantially different for FP programs than for other programs, though it is true that the functional nature of FP tends to reduce design bugs (and improve the localization of coding bugs) by eliminating side effects -- there is no way procedure X can affect the data managed by module Y, unless the two are explicitly connected. In other words, the result of a correct function will be correct as long as its input is correct, no matter what other functions are incorrect.

In addition to the -d, -e and -t switches described above, fpc provides stack dumps and

checkpoints. A stack dump occurs whenever a condition is encountered that would lead to bottom: for instance, when the wrong type of argument is passed to a primitive, e.g. if an atom is given to an apply-to-all expression or a non-numeric value is used as the input to iota. The stack dump is interactive if the input to the function came from a terminal, and placed in the file **core** otherwise. Here is a sample program and the stack dump it caused:

```
Def filter null o 2 -> _<>;
          (bu = 0) o mod o [1 o 2, 1] -> filter o [1, tl o 2];
# notice: the bug is in the following line. filter should be
# composed with [1, tl o 2] instead of [2, tl o 2]
          apndl o [1 o 2, filter o [2, tl o 2]
Def sieve null -> id; apndl o [1, sieve o filter o [1, tl]
Def primes sieve o tl o iota

% primes
10
error: bottom produced during execution
mod: second argument is not a number
<4, <3, 4, 5, 6, 7, 8, 9, 10>>
do you wish a stack dump (y/n)?
y
interactive stack dump?
y
dumping the relevant portions of the stack:
called by routine filter, with input
<<3, 4, 5, 6, 7, 8, 9, 10>, <4, 5, 6, 7, 8, 9, 10>>
continue stack dump?

called by routine filter, with input
<2, <3, 4, 5, 6, 7, 8, 9, 10>>
continue stack dump?

called by routine sieve, with input
<2, 3, 4, 5, 6, 7, 8, 9, 10>
continue stack dump?
n

aborting...
%
```

As before, we use a special script to distinguish what the computer typed from what you had to type. As you can see, a stack dump is fairly self-explanatory. Typing <return> to most questions will invoke the default option, as in "continue stack dump?", where the default option is yes. The error that originally caused bottom appears at the very top: the primitive function mod expects a pair of integers as its argument, and the second element of the list was a vector intead of a number. The listing can be aborted at any time, or it can be made non-interactive by answering the first question with 'n<return>'.

A checkpoint is a use of the primitive function checkpoint. The function prints its input onto the error output and lets the user continue the computation, abort the computation, or display the current call stack. The checkpoint function then returns

its input, so checkpoint may be inserted painlessly anywhere in an existing composition to display the flow of data.

When an incorrect output is produced, the location of the bug can usually be isolated by tracing. Compiling with -d or -t, the inputs and outputs of each function called can be monitored; at some point in the trace, one of the functions will produce an unexpected output for a correct input; that function is the one that contains the bug. The trace, or the stack dump, or both, may be used to isolate the function or functions that contain the bug, i.e., that produce the wrong output given the correct input. Notice how this is much easier than debugging a conventional program -- there is no need to examine individual variables or contents of files; the flow of data is entirely explicit and visible at all times.

Once you have identified the function that has the bug, you may still be mystified as to just why the function doesn't produce the right result. This is where you should use checkpoint. Usually checkpoint is composed between one expression and another to view intermediate results. As soon as you see an intermediate result that doesn't match your expectations, you have restricted the location of your bug. Once you have fixed the bug, remember to take out all checkpoints! They do not belong in production programs.

If you ever get a system error (segmentation fault, bus error, disk overflow, division by 0), either you have compiled with -n or linked to nocheckfp or both, or the fault lies with FPC. In the first case just recompile without the -n switch and link to fp.o, and run your program again to see where the problem lies. You should get an error message and be able to get a stack dump. In the second case you need to get in touch with your FPC maintainer or notify me (the author of this report), since it should be impossible to write an FP program that produces a system error. Bottom as described in [Ba 78] is detected and reported to the caller of the program in every case other than nontermination, i.e., infinite recursion or endless while loop.

6 The FP programming language

This section describes all of the primitives provided by FPC that are not described in section 7; section 7 describes the nonfunctional primitives provided by FPC.

The primitives not described in [Ba 78] are marked by a dot (•); they have been found sufficiently useful that they were included in FPC. We first describe the data types of FP, then document all the primitive functions, then the functional forms. Note that comments extend from any # (hash mark, pound sign) to the end of the line they're on; #'s in string or character constants do not mark the beginning of comments. Also note that the case (upper/lower) of words is significant: Def, for instance, must always be written with an uppercase 'D' and lowercase 'ef'.

FP data types

Objects in FP belong in one of the following classes:

<>: nil is the vector of length 0, is an atom. It is the only object for which *null* returns T.

T, F: the boolean values are returned by the predicate part of conditionals and while loops, are given to boolean operators, are returned by relational and boolean operators, and are atoms.

0, 1, -1, 2, -2, ...: the integers are given to and returned by arithmetic operators and iota. Both integer and floating-point numbers are atoms.

0.001e+5: • the floating-point numbers are treated exactly like integers, except that any operation performed on a mixture of floating-point and integer numbers returns a floating point number; *trunc* returns the integral part of any floating point number.

symbol: symbolic atoms may be entered as any sequence of letters or digits beginning with a letter. The function implode lets you define symbols whose name includes arbitrary characters.

'x: • characters are usually part of some string. Characters are atoms.

"xyz", or **<'x, 'y, 'z>**: • strings (the two representations are equivalent; the first one is the one used for output) are vectors of characters. *Implode* takes a string and produces the symbolic atom whose name is the given string; explode takes an atom and returns the string corresponding to the atom name. Strings are vectors, not atomic objects.

<1, x, "abc", <>, <hello, 1.2>>: vectors are ordered collection of any number of objects of any type. Most primitives in FP operate on vectors, particularly on vectors of length 2, called pairs. **<hello, 1.2>** is a pair. Vectors are the only mechanism for building complex objects in FP -- they can be used in place of the arrays, records, or lists of other languages. Vectors are not atomic objects.

Primitive functions

+, -, *, div, mod are the binary arithmetic operators; they accept pairs of numbers and return numbers. **div** applied to two integers always returns the number less than or equal to the exact quotient. **mod** may only be applied to pairs of integers, and returns the positive remainder of dividing the first number by the second. Examples: + applied to <2, 3> returns 5; - applied to <7.0, 9> returns -2.0; * applied to <7, -1> returns -7; **div** applied to <9, 4> returns 2; **mod** applied to <9, 4> returns 1; mod applied to <-3, 2> returns 1.

=, != are the equality operators; they accept pairs and return whether the two elements of the pair are equal or unequal, respectively. = and != always return the opposite value. Examples: = applied to <<<1>>, <<1>>> returns T, applied to <<<1>>, <1>> returns F, != applied to the same examples returns F and T, respectively.

>, <, >=, <= • are the relational operators; they accept pairs of numbers, atoms or characters and return their relation. Any number is less than any atom; any atom is less than any character. Atoms and characters are ordered in a system-dependent way.

and accepts a pair of boolean values; it returns T if its input is <T, T>, F otherwise.

apndl, apndr append a new element to the left or to the right of a vector, respectively. Examples: **apndl** applied to <1, <2, 3, 4>> returns <1, 2, 3, 4>; **apndr** applied to <<1, 2, 3>, 4> returns <1, 2, 3, 4>.

append • concatenates all the vectors that are the elements of its input vector, discarding any top-level nils. Example: **append** applied to <<1, 2> <> <<3, 4> 5>> returns <1, 2, <3, 4>, 5>.

atom returns a boolean value indicating whether the input was atomic or not. Atom returns T for nil, booleans, numbers and characters; returns F for vectors and strings.

checkpoint • accepts any input and returns it (like id); in addition, it prints its input to the error output (usually the screen) and lets the user do a stack dump or abort the computation. It should only be used for debugging purposes.

distl, distr take a pair consisting of any object and a vector, and return a new vector of the same size as the vector in the input, where each element consists of the pair made up by the object and the corresponding element of the input vector. The object is on the left in distl and on the right for distr, both for the input and output values. Examples: **distl** applied to <a, <1, 2, 3>> returns <<a, 1>, <a, 2>, <a, 3>>, **distr** applied to <<a, b, c> <1, 2>> returns <<a, <1, 2>>, <b, <1, 2>>, <c, <1, 2>>>.

error • takes any input, prints it on the standard error output (normally the screen) and returns bottom, i.e., generates a stack dump. If the program was linked to nocheckfp.o, **error** aborts the program after printing its value.

explode • accepts as input a symbolic atom and returns the string corresponding to the atom's name. Example: **explode** applied to AnyAtom returns "AnyAtom".

id accepts any input and returns it. It is useful as a place-holder for the argument in constructors.

implode • accepts as input a string and returns the symbolic atom having as name the given string. The string may contain arbitrary characters, which could cause confusion when the atom is printed out. Implode is not extremely useful, but its provided because it is the inverse function of explode and is occasionally useful in parsing strings. Example: **implode** applied to "abc123" returns the atom abc123.

iota accepts an integer greater than or equal to 0 and returns a vector of integers from 1 to that number. Iota returns nil when given 0, <1> when given 1. Example: **iota** applied to 5 returns <1, 2, 3, 4, 5>.

length accepts a vector or nil and returns the number of elements in that vector. Examples: **length** applied to <> returns 0; **length** applied to <<a, b>, c, <d, <e, f, g>>, h> returns 4.

neg • accepts any number and returns its negation. Examples: **neg** applied to -7 returns 7; **neg** applied to 3.14 returns -3.14; **neg** applied to 0 returns 0.

newline • is a constant function: it accepts any input and returns the string that on the current system is used to signal the end of a line of text. This string may be appended in any arbitrary string to signal a line break for pretty-printing output. A string is returned (as opposed to a single character) since some systems may use a string of characters to signal a new line.

not takes as input a boolean value and returns the other boolean value.

null takes any input and returns whether it is the empty vector.

or accepts a pair of boolean values; it returns F if its input is <F, F>, T otherwise.

reverse accepts a possibly empty vector and returns a vector of the same length where the left to right order of the elements has been reversed. Example: **reverse** applied to <<a, b>, 2, 3, 4, <c, d, e>> returns <<c, d, e>, 4, 3, 2, <a, b>>.

rotl, rotr accept as input a nonempty vector and return the same vector with the leftmost element moved to the rightmost position (**rotl**), or with the rightmost element moved to the leftmost position (**rotr**). Examples: **rotl** applied to <1, 2, 3, 4> returns <2, 3, 4, 1>; **rotr** applied to <a, b, c, d, e> returns <e, a, b, c, d>; either primitive reverses the order of the elements of any pair it is applied to.

tl, tlr accept a nonempty vector and return the same vector minus its leftmost (for **tl**) or rightmost element (for **tlr**). Examples: **tl** applied to <a, b, c> returns <b, c>; **tlr** applied to the same returns <a, b>; either applied to <a> returns <>.

trans accept a vector of vectors; the elements of the input must be either all nil or all the same length. The result is the transpose of the input, a vector with the same length

as each element of the input, and where the first element contains all the first elements of the vectors in the input, in order, the second element contains all the second elements of the vectors in the input, and so on, until the last element contains all the last elements of the vectors in the input. The transpose of a vector of nils is nil. Example: **trans** applied to

```
<<1,   2,      3,          4           >,
 <a,   b,      c,          d           >,
 <<1>, <1, 2>, <1, 2, 3>, <1, 2, 3, 4>>>
```

returns

```
<<1, a, <1>          >,
 <2, b, <1, 2>       >,
 <3, c, <1, 2, 3>    >,
 <4, d, <1, 2, 3, 4>>>.
```

trunc • takes as input a floating-point number and returns the largest integer less than or equal to the input. Example: **trunc** applied to -5.2 returns -6.

Primitive functional forms

o: composition takes as parameters two or more functional expressions; takes as input any value, and returns the result of applying the rightmost expression to the value, the expression to its left to the result of that, and so on until the leftmost expression has been applied to the result of the expression to its right; the result of the leftmost expression is the result of the composition. Example **iota o length o apndl** applied to <<a, b, c, d>, e> returns <1, 2, 3, 4, 5>.

n: a selector (**n** stands for any positive number greater than 0) takes as input a vector of length at least **n** and returns the **n**th element of that vector (counting from the left). Examples: **2** applied to <a, b, c, d> returns b; **7** applied to <p, q, r, s, t, u, v> returns v; **1** applied to <<x, y, z>> returns <x, y, z>.

nr: a right selector selector (**n** stands for any positive number greater than 0) takes as input a vector of length at least **n** and returns the **n**th element of that vector (counting from the right). Examples: **2r** applied to <a, b, c, d> returns c; **7r** applied to <p, q, r, s, t, u, v> returns p; **1r** applied to < <x, y, z>> returns <x, y, z>.

pred -> then ; else: a conditional takes as parameters three functional expressions; it takes as input any value, applies the first functional expression to it, and if the result of the application is true, returns the result of applying the second expression to the input; if the result is false, conditional returns the result of applying the third expression to the input. Note that since every FP expression is required to return some value, the else part is must be present, otherwise no value could be returned if the predicate returned false. If the else part should never occur, use the primitive function *error* to report any problem to the user of the program. Example: **null -> _0; atom -> id; length** applied to 3 returns 3; applied to <> returns 0; applied to

<a, b, c, d> returns 4.

aa: apply-to-all takes as parameter a functional expression; takes as input a possibly empty vector, and produces as output a vector of the same length as the input but in which every element has been replaced by the result of applying the functional expression to the corresponding element of the input. The elements may be processed in any order. Example: **aa neg** applied to <1, 7, -2, 5, 4, 0> returns <-1, -7, 2, -5, 4, 0>.

\V: • tree insert (also known as tree) takes as parameter a functional expression; takes as input a nonempty vector, and produces as output the result of combining all the elements of the vector pairwise using the functional expression. If the input is a vector with a single element, that element is returned. Any odd elements of a vector will be combined with a previous result instead of with an element; results will then be combined pairwise using the functional expression. The odd elements may be taken from the beginning of the vector, from the end, or from anywhere else. Tree insert is generally faster than either left-insert or right-insert when using functions that are both associative and commutative, such as +, *, merge, and is only very rarely slower. Note that fpc optimizes the common cases \V+, \V*, \V**and**, and \V**or** to be particularly efficient (as long as -d, -e are not used). Examples of tree insert: \V**id** applied to <1 2 3 4 5 6 7> can return <<<1, 2>, <3, 4>>, <<5, 6>, 7>>, but also <<1, <2, 3>>, <<4, 5>, <6, 7>>>, or <<<1, 2>, 3>, <<4, 5>, <6, 7>>>. \V+ applied to <1, 2, 3, 4, 5, 6, 7> always returns 28.

/: insert (also known as left insert or insert-from-left) is the insert described in [Ba 78]. It takes as parameter a functional expression; it takes as input a nonempty vector and for a vector of length one returns the single element of the vector. For a longer vector, insert returns the result of applying the functional expression to the pair consisting of the leftmost element and the result of applying itself to the tail of the vector (this is a recursive definition). Example: /**id** applied to <1, 2, 3, 4, 5> returns **id** applied to (<1, /**id** applied to <2, 3, 4, 5>>), which gives <1, <2, <3, <4, 5>>>>.

\: • right insert (also known as insert-from-right) takes as parameter a functional expression; it takes as input a nonempty vector and for a vector of length one returns the single element of the vector. For a longer vector, right insert returns the result of applying the functional expression to the pair consisting of the result of applying itself to the right tail of the vector and the rightmost element of the vector (this is a recursive definition). Example: **id** applied to <1, 2, 3, 4, 5> returns **id** applied to <(**id** applied to <1, 2, 3, 4>), 5>, or <<<<1, 2>, 3>, 4>, 5>.

[f1, f2, .., fn]: construct takes as parameters 0 or more functional expressions; takes any input; returns nil when no functional expressions are given, as in []; returns the vector with elements given by the results of applying the functional expressions to the input otherwise. The results are in the same order as the functional expressions; however, the functional expressions may be executed in any order. Example: [\V+, **length**] applied to <1 2 3 4> returns <10, 4>.

_obj: constant takes as parameter an object; takes as input any value and returns the parameter. Examples: **_3** always returns 3; **_<a, b, c>** always returns <a, b, c>; **_<>** always returns nil; **_a** always returns the atom a; **_'a** always returns the character a; **_"abcd"** always returns the string "abcd", which is the same as the vector of characters <'a, 'b, 'c, 'd>.

bu fun obj: bu (binary-to-unary) takes as parameters an expression and an object; takes any input, pairs it with the object parameter, and returns the result of applying the expression parameter to the pair of the object parameter and the input.Example: **bu apndr <1, 2, 3>** applied to 4 returns <1, 2, 3, 4>.

bur fun obj: bur (binary-to-unary on the right) takes as parameters an expression and an object; takes any input, pairs it with the object parameter, and returns the result of applying the expression parameter to the pair of input and the object parameter.Examples: **bur - 6** applied to 17 returns 11; **bur apndr 4** applied to <1, 2, 3> returns <1, 2, 3, 4>.

while pred iter: while takes as parameters two expressions; takes any input, and applies its first parameter to it. If the predicate is false, the input value is returned; otherwise it applies its second expression to the input, and starts again using the result of that application as its input. Example: **while not o atom 1** always returns the first atomic element in an arbitrarily nested structure, or its argument if the argument is atomic.

7 FPC specials

This section describes non-standard routines that allow a limited form of input/output in FP programs, above and beyond that provided by the -o, -i and -a switches. These functions should be avoided whenever possible and are experimental, in the sense that they may be removed from future versions of FPC; I appreciate feedback on their usefulness for actual programming. When using these functions, remember that the execution order of the subexpressions of many of the functional forms is not defined and may change from one implementation to another or even within a given implementation.

arguments ignores its input and always returns the same value in any given invocation of a program. Argument returns a vector of strings corresponding to the arguments given in the call of the program. If any options were given, the option is returned as a pair: the first element is the option character, the second one is nil if the option did not have a parameter, or a string containing the parameter if a parameter was defined. Example: if the progam was called (under unix) as "program arg1 -q arg2 -pxyz", a call to **arguments** would return <"arg1", <'q, <>>, "arg2", <'p, "xyz">>.

filetype accepts a string representing a file name and returns one of the atoms *none*, *empty*, *data*, *text*, *binary*. A file is of type *none* if it does not exist; it is *empty* if it exists but is of size 0; it is of type *data* if it can be parsed by the FP reader; it is of type *text* if it contains only printable and formatting characters (including line separators); it is of type *binary* otherwise. Notice that files of type *data* are logically also of type *text* (as long as they don't contain binary data after the end of the FP object), but *data* is returned anyway.

input accepts a string representing a file name and returns a string containing all the characters of the file if the file is a text file; **input** fails (becomes bottom) if the file does not exist or contains non-textual characters. Example: if the file "ex1" contains the text

```
the quick brown fox jumps
over the lazy dog
```

input applied to "ex1" would return "the quick brown fox jumps<newline>over the lazy dog", where <newline> stands for the system's newline string.

read is like **input**, but parses the file and returns the FP object corresponding to the string contints, instead of the string of characters found in the file. Example: if the file "ex2" contains

```
<this, is, a, data,
file>
with some comments at the end.
```

read applied to "ex2" would return <this, is, a, data, file>, where **input** would return the string "<this, is, a, data,<newline>file><newline>with some comments at the end.<newline>".

trace is functionally equivalent to **id**. It prints its input, which must be a (possibly empty) string, to the stderr output, without surrounding quotes.

8 How does FPC ...?

This section briefly describes how FPC implements the FP language. For more details, write short FP programs and look at the resulting C code, or check the comments in the source of the FPC translator.

The basic principle of FPC is that functional forms produce in-line code, wheereasprimitives are implemented in the run-time library. The run-time library, fp.c, is used to generate (via different compiler switches) all three of fp.o, nocheckfp.o and debugfp.o, so should be modified with that in mind. Memory

management is by reference counting: FP does not allow the creation of self-referential structures, so reference counting is possible.

For the specific implementations of the functional forms and primitives, go ahead and write short FP programs and look at the resulting code, or study fp.c. In particular, look at the difference in the generated code when error checking is used (the normal case) and when it is turned off (-n); the code will be a lot clearer when error checking is turned off. FPC tries to produce C code that is legibly indented; try using the Unix *indent* program if you prefer a different style of indentation. In fp.c notice that NOCHECK is defined when compiling nocheckfp, and DEBUG is defined when compiling debugfp.

All FP objects are of type fp_data; fp_data is a pointer to a cell in memory containing a type (fp_type), a reference count (fp_ref), and the object itself. In the case of atomic objects, the value of the object is stored in the cell itself, except for symbolic atoms which store a pointer to the atom name; for vectors, the cell stores a pointer to the vector's leftmost element (fp_entry) and a pointer to the rest (**tl**) of the vector (fp_next). fp_next of the cell pointing to the last element of a vector is the null pointer, i.e. has the value 0. Unlike the definition of FP, this implementation is asymmetric, since it allows faster access to the left end of a vector than to the right end; hopefully most users will not notice the difference. In several tests, this implementation was usually much faster, and never substantially slower, than a previous implementation which used dynamically-allocated arrays to implement vectors.

The run-time library exports the memory management functions newconst, newcell, newpair, newvect, returnvect. The new.... functions are split since FPC usually knows which one to generate and the generated code runs faster if the decision does not have to be made at run time. returnvect is called when an object's reference count reaches 0, but only on cells which are part of a vector -- constant cells are never returned (the assumption being that they are used over and over again, and not created very often).

The reference counting goes as follows: each functional form and primitive assumes that the data it is given as input has at least one reference to it (i.e., its own reference to it), and that the reference counter reflects that. The reference counter is decremented if and only if the input is not part of the result. If the reference counter is 0 and the object is a vector, returnvect is called to deallocate the cell and recursively decrement the reference counts of the element and the tail of the vector.

Notice a slight optimization could be obtained by having primitives such as distl check whether their input has only one reference to it and, if so, re-using the existing backbone. This is not done at this time.

Any program compiled with the -s switch and any program for which the allocation and deallocation counts are not the same print, after their normal output, the number

of cells they allocated to vectors, the number of cells that were returned, and the maximum amount of space used. If the number of cells allocated and returned was not the same, there was some error in the reference counting. Reference counting is very sensitive to changes, so if the error is reported, any recent changes in the code generation or fp.c should be scrutinized for possible errors. It is also possible to compile fp.c (by defining CHECKREF) so that the print routine will print the reference count of all the objects that are printed out.

9 Installing FPC

To install fpc, you need to decide what directories the binaries will reside in, and make sure you have write permission to them.

If you're on a Unix system, the only things you should need to do is edit fp.h to adjust MAXINT to be your C compiler's maximum integer, adjust the makefile so BIN and LIB are the directories in which you want to see the binaries and runtime system, and run make. You have to edit the makefile of the directory 'lib' and once again run make. Once you are convinced that the program runs correctly, you can run 'make release' in both directories. I generally use the file 'prims.fp' to test the primitives: any serious problem will often show up when running prims.

If you are not on a Unix system and make is not available, you may have to do the last step by hand. If you have lex and yacc, run them on fpg.l and fpg.y, respectively, to produce lex.yy.c and y.tab.c; you may want to increase the size of the yacc stack in y.tab.c to 8192 instead of the standard 128, so that large compounded conditional expressions do not overflow the yacc stack. If you do not have lex and yacc available, use lex.yy.c and y.tab.c as supplied. Compile all the .c files, then compile fp.c to nocheckfp.o after defining NOCHECK, and to debugfp.o after defining DEBUG. Then link fpc.o, code.o, expr.o, parse.o and y.tab.o together into an executable named fpc, and you are done. If you are installing libraries, you can install fp.o in the library libfp.a (I'm using the UNIX naming conventions, you will have to translate them to your own), nocheckfp.o in libnfp.a, and debugfp.o in libdfp.a.

10 The outlook for FP

FP had its moment of glory back in 1978 and is now slowly losing ground to lazy functional languages (FP is strict, i.e. each argument to a function must be completely evaluated before the function can be evaluated) and to improvements on FP itself. To the latter belong extended FP [Ba 81] and the not-yet-completed programming language FL [BWW 86]. The former includes a host of languages such as KRC, sugar, Miranda. So why should anyone bother to program in FP instead of one of these other languages?

There are several answers. First of all note that a programming language's life cycle

is quite long: FORTRAN, APL and LISP are approaching 30 years of age, and show no sign of disappearing. Similarly, even though FP is no longer as fashionable as when it first came out, more and more implementations are being produced; the language is quite popular and possibly becoming more so. In fact, it is just now that useful and reasonably efficient implementations (such as FPC) are becoming available. This is not (yet) the case for FP's competitors. In addition, FP is in general a subset of its successors, so any program written in FP can usually be converted to run in FL or extended FP without too much effort.

One reason for programming in FP instead of in a lazy language is speed -- FP programs can compile to more efficient code than most equivalent lazy programs, and can in addition run much faster on any parallel implementation, since all the arguments to a function are evaluated at the same time; for a lazy language, the arguments would not be evaluated until actually used, which means that very often only a few parts of the argument can be evaluated in parallel.

Finally, FP is, in its own way, a very elegant language -- the lack of variables, which makes it somewhat unreadable to beginners, also makes it very compact and clean.

There are also many reasons for NOT using FP. If your program does interleaved reads and writes of files, or if you think you need higher order functions or lazy evaluation, or if you need the high uniprocessor efficiency provided by an imperative language, you should probably not be using FP. In all other cases, FP is recommended!

Finally, I would like to thank my advisor and the people at UNC who have offered help and encouragement throughout this project: Gyula Mago', Brad Bennett, Vernon Chi, Lakshmi Dasari, Bill Gibson, Tai-Sook Han, Bharat Jayaraman, David Middleton, Will Partain, Raj Singh, Bruce Smith, and Don Stanat.

References

[Ba 87] S. Baden, Berkeley FP User's Manual, Rev. 4.1, September 29, 1987

[Ba 78] John Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs", Turing Award lecture, Communications of the ACM, (21, 8), August 1978, pp. 613-641

[Ba 81] John Backus, "The Algebra of FUnctional Programs: Function Level Reasoning, Linear Equations, and Extended Definitions", pp. 1-43 of "Formalization of Programming Concepts. Proceedings of an International Colloquium" (Apr 19-25, 1981), edited by J. Diaz and I. Ramos, Springer Verlag, Lecture Notes in Computer Science, Volume 107.

[BWW 86] John Backus, John Williams and Edward Wimmers, "FL Language

Manual (Preliminary Version)", IBM Almaden Research Center Research Report, San Jose, CA, Nov. 7, 1986.

[GD 83] Adele Goldberg, David Robson, "Smalltalk-80, the language and its implementation", Addison-Wesley, 1983

[HHH 86] Tien Huynh, Brent Hailpern, Lee W. Hoevel, "An execution architecture for FP", IBM Journal of Research and Development, (30, 6), November 1986, pp. 609-616

[Ro 87] Arch D. Robison, "Illinois Functional Programming: A Tutorial", BYTE (12, 2), February 1987, pp. 115-125

Appendix A: some examples

Compute the arithmetic average of a vector of numbers:

```
Def ave div o [\/+, length]
```

An efficient fibonacci program:

```
Def fib (bu > 2) -> id; fib2 o (bur apndl <1, 0>)
Def fib2 (bu = 0) o 1 -> 2; fib2 o [(bur - 1) o 1, + o tl, 2]
```

Produce a list consisting of all the atoms in the argument, in order:

```
Def flatten null -> id; atom -> [id]; append o aa flatten
```

Quicksort in FP:

```
Def before append o aa ( > -> tl; _<>)
Def same   append o aa ( = -> tl; _<>)
Def after  append o aa ( < -> tl; _<>)
Def qsort null -> id;
          append o [qsort o before, same, qsort o after] o
             distl o [1, id]
```

Generate the list of prime numbers between 1 and the input:

```
Def primes sieve o tl o iota
Def sieve null -> id;
          apndl o [1, sieve o filter o [1, tl]]
Def filter null o 2 -> id;
           (bu = 0) o mod o [1 o 2, 1] -> filter o [1, tl o 2];
           apndl o [1 o 2, filter o [1, tl o 2]]
```

Inner product and matrix multiplication:

```
Def IP /+ o aa * o trans
Def MM aa aa IP o aa distl o distr o [1, trans o 2]
```

Some utilities to implement a stack storage:

```
# a stack is accessed through newstack, push, pop, top, isempty
Def newstack _<>
Def isempty null
Def top 1
Def pop tl
Def push apndl
```

A function to convert integers to strings:

```
Def inttostr (bur < 0) o 1 ->
                    (bu apndl '- ) o inttostring o neg;
              aa printdigit o reverse o makedigit
```

```
Def makedigits (bur < 10) -> [id];
               apndl o
               [(bur mod 10), makedigits o [(bur div 10), 2]]
Def printdigit 1 o (bur sel_n "0123456789") o [(bu + 1], _1]
Def sel_n left_n o [2 o 1, right_n o [- o 1 o 1, _1], 2]]
Def left_n append o aa (< o [1, 1 o 2] -> _<>; [2 o 2]), o
          distl o [1, pairpos o 2]
Def right_n append o aa (>= o [1, 1 o 2] -> _<>; [2 o 2]), o
           distl o [1, pairpos o 2]
Def pairpos null -> _<>; trans o [iota o length, id]
```

A library to implement tables (stores):

```
# A store is a place to keep objects in and retrieve them by key.
# A key is an atom or a number.
# newstore: x => a new (empty) store
Def newstore _<>
# store: <<key, value> store> => new store
Def store apndl o [1, unstore o [1 o 1, 2]]
# retrieve: <key store> => value if any, <> otherwise
Def retrieve (null -> id; 1) o append o
             aa (= o [1, 1 o 2] -> tl o 2; _<>) o distl
# unstore: <key store> => new store
Def unstore append o aa (= o [1, 1 o 2] -> _<>; id) o distl
# storesize: store => number of items in the store
Def storesize length
# allstored: store => vector of pairs with all keys and values
Def allstored id
```

An alternate (recursive) implementation of the above routines would use a tree to store the values:

```
# A tree is either nil or [key value left right], with left
# and right both being trees.
Def newstore _<>
Def store null o 2 -> [1 o 1, 2 o 1, _<>, _<>];
# at leaf, insert the value.
         < o [1 o 1, 1 o 2] ->
# desired key is less than node key, insert at left
            [1 o 2, 2 o 2, store o [1, 3 o 2], 4 o 2];
         > o [1 o 1, 1 o 2] ->
# desired key is greater than node key, insert at right.
            [1 o 2, 2 o 2, 3 o 2, store o 1, 4 o 2]
# else keys are equal, replace the value field
         [1 o 2, 2 o 1, 3 o 2, 4 o 2];
Def retrieve null o 2 -> 2;      # at leaf, key not found
```

```
            < o [1, 1 o 2] -> retrieve o [1, 3 o 2];
            > o [1, 1 o 2] -> retrieve o [1, 4 o 2];
            [1, 2] o 2          # keys =, key found
Def unstore null o 2 -> id;     # at leaf, key not found
           (< o [1, 2] -> [2, 3, unstore o [1, 4], 5];
            > o [1, 2] -> [2, 3, 4, unstore o [1, 5]];
            unstorelift o tl] o apndl
# unstorelift: store => store where the root has been replaced
# by its left child, recursively.
Def unstorelift null o 3 -> 4;
                [1 o 3, 2 o 3, unstorelift o 3, 4]
Def storesize length o allstored
Def allstored null -> id;
             apndl o [[1, 2], append o aa allstored o [3, 4]
```

A package of set operations on lists:

```
# member: <item, vector> => boolean
Def member null o 2 -> _F; \/or o aa = o distl
# include: <item, vector> => new vector, where item is apndl'd
# to vector if and only if it was not previously a member.
Def include member -> 2; apndl
# exclude: <item, vector> => vector where any elements that
# were equal to item have been removed
Def exclude null o 2 -> 2; append o aa (!= -> tl ; _<>) o distl
# index: <item, vector> => (one) position of item in the set, or 0
Def index null o 2 -> _0;
          \/ ((bu = 0) o 1 -> 2; 1) o aa (= o 2 -> 1; _0) o
# for input <q, <x1,..xn>> we pass up <<1, <q, x1>>,..<n, <q, xn>>>
          trans o [iota o length, id] o distl
```

Appendix B: library functions

Besides the primitive functions, fpc comes with a set of library functions which seem to be useful for general programming. They are defined in FP, and in fact their definition should be on line somewhere on your system. They are loaded automatically by specifying -lfp. A synopsis follows. You will recognize some fo the functions from Appendix A.

Library lib.fp:

pairpos: $\langle x_1, \ldots x_n \rangle$ => $\langle\langle 1, x_1\rangle, \ldots \langle n, x_n\rangle\rangle$

allpairs: $\langle x_1, x_2, \ldots x_n \rangle$ => $\langle\langle\langle\rangle, x_1\rangle, \langle x_1, x_2\rangle, \ldots \langle x_{n-1}, x_n\rangle, \langle x_n, \langle\rangle\rangle\rangle$

ntl: $\langle n, \langle x_1, \ldots x_m\rangle\rangle$ => $\langle x_{n+1}, \ldots x_m\rangle$

nhd: $\langle n, \langle x_1, \ldots x_m\rangle\rangle$ => $\langle x_1, \ldots x_n\rangle$

seln: $\langle\langle s, l\rangle, \langle x_1, \ldots x_n\rangle\rangle$ => $\langle x_s, \ldots x_{s+l-1}\rangle$

selectl: $\langle i, \langle x_1, \ldots x_n\rangle\rangle$ => x_i

selectr: $\langle i, \langle x_1, \ldots x_n\rangle\rangle$ => x_{n-i+1}

breakup: $\langle\langle 1, i_2, \ldots i_n\rangle, \langle x_1, \ldots x_m\rangle\rangle$ => $\langle\langle x_1, \ldots x_{i2-1}\rangle, \ldots . \langle x_{in}, \ldots xm\rangle\rangle$

permute: $\langle\langle i_1, x_{i1}\rangle, \ldots \langle i_n, x_{in}\rangle\rangle$ => $\langle x_1, \ldots x_n\rangle$, as long as the set of i_j's forms a permutation of the integers between 1 and n.

rank: $\langle x, \langle x_1, \ldots x_n\rangle\rangle$ => m, the number of x_i's <= x

Library store.fp

Implements a table or store.

newstore: x => empty store

store: <<key, value>, store> => store with the new entry

retrieve: <key, store> => <key, value> or <>

unstore: <key, store> => store without the given entry

storesize: store => n, the number of entries in the store

allstored: store => $\langle\langle key_1, value_1\rangle, \ldots \langle key_n, value_n\rangle\rangle$ (the order is arbitrary)

haskey: <key, store> => T or F

Library set.fp

Defines set operations on vectors.

member: <item, vector> => T or F

include: <item, vector> => member -> apndl; 2

exclude: <item, vector> => not o member -> 2; "vector - item"

includem: $\langle\langle item_1, \ldots item_n\rangle$, vector> includes all items

excludem: $\langle\langle item_1, \ldots item_n\rangle$, vector> excludes all items

index: <item, vector> => the (first) selector for item in the vector, or 0

Library format.fp

Defines conversion to and from string (printable) representation.
fpformat: <obj_1, ... obj_n> => string, where each of the objects is a string or any atom.
fpscan: <<$form_1$, ... $form_n$>, string> => <obj_1, ... obj_m>, where form is either one of the symbols: symbol, number, integer, float, boolean, character, string, or a character or a string which must be matched exactly. $m \leq n$ if not all of the string could be matched.
symbol: x => T if x is any atomic symbol, F otherwise
number: x => T if x is any number, F otherwise
character: x => T if x is any character, F otherwise
boolean: x => T if x is T or F, F otherwise
vector: x => T if x is nil or any vector, F otherwise
string: x => T if x is nil or any vector made up of only characters, F otherwise
inttostring: <number, base> => "number in base"
charalpha: char => T if char is an alphanumeric character, F otherwise
charupper: char => T if char is an uppercase alphanumeric character, F otherwise
charlower: char => T if char is a lowercase alphanumeric character, F otherwise
chardigit: char => T if char is a digit ('0..'9) character, F otherwise
charoctdigit: char => T if char is an octal digit ('0..'7), F otherwise
charhexdigit: char => T if char is a hexadicimal digit ('0..'9, 'a..'f, 'A..'F), F otherwise
charspace: char => T if char is an alphanumeric character, F otherwise

Appendix C: the FP grammar

```
FPProgram ::= FPDef | FPProgram FPDef.
FPDef ::= Def Symbol Toplev
Toplev ::= Comp -> Then ; Else      |
           bu Toplev Object         |
           bur Toplev Object        |
           while Toplev Toplev      |
           Comp.
Comp ::= Expr | Expr o Comp.
Expr ::= ( Toplev ) | aa Expr | [] | [ToplevList] |
         / Expr | \ Expr | \/ Expr | _Object | Sel | Rsel |
         Symbol | + | - | * | = | < | > | >= | <= | !=.
ToplevList ::= Toplev | Toplevlist , Toplev.
Object ::= T | F | [-] Sel | Symbol | String | Char | Float |
           <> | < ObjList >.
ObjList ::= Object | Object , ObjList.

Comment ::= # Chars EndOfLine.
```

Appendix D: modifying FPC

FPC is divided into several modules, as shown in figure 1:

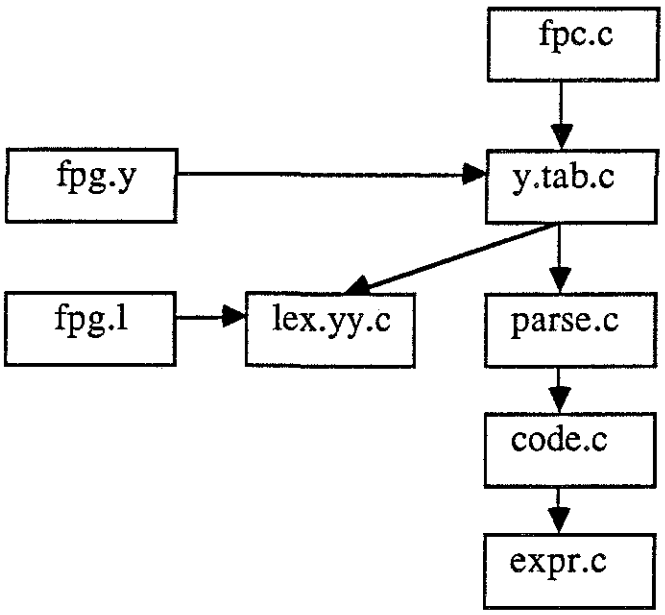


Figure 1: the module structure of FPC

The main module, fpc.c, interprets the options (and sets global variables accordingly), then calls the parser. The parser calls parse.c, which builds a parse tree from the program. Parse.c then calls code.c, which does some program transformations and simple optimizations, and generates the stubs for the C procedures. Then code.c calls expr.c, which generates the code for the individual functional expressions. expr.c does a recursive tree traversal of the tree built up by parse.c.

There are a number of global switches, defined in fpc.h, which indicate whether the module being compiled should be debugged, whether error checking should occur, and so on. These switches are checked as appropriate when generating code. A good strategy for reading the code generator is to assign a given value to all the switches and then mask out the parts that are not selected for, then start again with a different value (e.g., with check = 1 or check = 0, and so on).

Each function is translated independently of all others; it is assumed that any function call is of the type fp_data function (), and takes a single argument of type fp_data. Two passes are made through (the parse tree of) each function, the first to determine which constants and how many variables are used, the second to generate the actual code. Variable allocation is frightfully inefficient: a new variable is used for almost every intermediate result. For a more efficient method of variable utilization see [HHH 86]. This more efficient scheme is not currently implemented in fpc.

In general, the only thing to really watch out for when modifying code generators is the reference counting. As long as you don't compile with -n or link to nocheckfp, the programs will keep track of the number of cells allocated and returned, and let you know if any discrepancy is detected. Normally an error in the reference counting will lead to a discrepancy. The only reasonable way I know of to find errors in the reference counting is to double-check any recent changes in fp.c or the code generators. Unreasonable ways include binary search in the FP program that causes the discrepancy (comment out half the program, run it, see if it has a problem, if not, comment out the other half and see what happens), and printing out all the intermediate results together with their reference counts (-d and debugfp.o).

fp.h defines the data structures used by the (translated) FP programs and fp.c. It includes stdfp.h which is the header generated for each translated FP program. In general, modifying a constant value (e.g., VECTOR) in stdfp.h will cause code.c to emit the new value in all newly translated programs.

Building the parse tree, which is defined in parse.h, is a reasonably straightforward application of lex and yacc. I have left much of the scaffolding in, so if you make a mistake you will probably get a message (when translating) of the form "compiler error ##". If you see it, find the error with that number (it should occur only once in the source) and you should be able to understand what the problem is from the context and any comments. The parse tree should be intuitively clear; if you have problems understanding it, check out the tree traversal routines in code.c and expr.c. For each function, we build a tree, then output the code for the function before parsing the next function.

The translator itself is also quite simple. fp.c is slightly less straightforward, partly due to the desire to have 3 different versions in one file (nocheckfp, fp, and debugfp), and partly due to some slight optimizations (such as splitting up the cell allocation functions for constants, vectors of length 1, and pairs from the more general case). The most complex function is transpose; next more complicated are filetype, the comparison routine (used by =, !=, >, <, >=, <=) and the arithmetic checking and setup routine. The input routine was written "by hand" and is reasonably straightforward. The output routine pretty-prints output and formats it so it (for reasonable nesting depths and identifier lengths) fits within an 80-column display.

In general, I find it fairly easy to modify fpc. If the modification is visible to the programmer, I usually document the change in the man page, fpc.1. If you are unfamiliar with nroff, just copy examples from elsewhere in the man page to achieve the desired effect. Last but not least, this technical report is subject to obsolescence, so if you find that something is different from the way it is described here, look for documentation (comments) in the code itself. Correspondingly, be sure to document any changes you make by adding appropriate comments both in the file headers and in-line with the changes. Your successors will be grateful to you.

Appendix E: the UNIX man page for FPC

NAME

fpc - fp to C compiler

SYNOPSIS

fpc [**options**] ... **file**

DESCRIPTION

FPC is an fp compiler. It produces as output C source files rather than object files. **FPC** accepts as arguments the names of the files to be compiled. Arguments of the form **name.fp** are taken to be fp source programs; each is compiled into the C source file **name.c.** Programs can be compiled as normal C source files; they must be loaded using one of the switches **-lfp, -lnfp, -ldfp.**

EXAMPLE

As an example, we compile mmult.fp and ip.fp. Assume the file mmult declares functions noop and matrixmult. The resulting program will execute the function matrixmult. It is assumed that ip.fp contains auxiliary functions needed by noop or matrixmult or both. To compile, do the following:

```
fpc -mmatrixmult mmult.fp ip.fp
cc -o mmult mmult.c ip.c -lfp
```

This will produce the file mmult which
accepts input from the user, and
applies matrixmult to it, and
outputs the result.

The first command above could have been entered as two separate commands:

```
fpc -mmatrixmult mmult.fp
fpc ip.fp
```

The two commands could have been given in either order.

OPTIONS

The following options are recognized by **FPC.** Notice that

options only apply to the first file name following the option.

-v (verbose) prints on the standard output the version number of the compiler and the names of the functions being compiled.

-d (debug) produces code to trace all function entries and exits and the arguments passed to them as well as the data they return. The trace is printed on the standard error output (stderr).

-e (entry/exit) same as -d, except that it does not print the arguments or the results of the functions.

-t**fun** (trace) like debug, but only for function **fun.**

-n (no check) the arguments to functional forms are not checked for correctness, and other optimizations are done which should speed up execution but make error detection and localization harder. It is expected that programs compiled with -n will be loaded with the library nfp.

-m
-m**fun** The compiler assumes that the first file name following the -m switch (call it **file.fp**) requires a **main** procedure. The **main** procedure

reads data from the standard input;
calls the `main' function on that data;
prints the result of the call.

If the second form is used, the `main' function is **fun;** for the first form, it is **file.**

-s (space) produces code to print out the maximum amount of space used by the program, as well as the number of cell allocations and de-allocations. Can only be used in conjunction with -m.

-i (string input) can only be used with -m. Specifies that the data obtained from the standard input be

passed to the main function as a string (a vector of characters) rather than as an ffp object. In other words, **abc** would be given to the `main' function as <**'a, 'b, 'c**> rather than as the atom **abc.**

-o (string output) can only be used with -m. Specifies that the data returned by the `main' function be output in raw form. The data may be a string or a vector of pairs <**filename,** string>, (e.g. <<file1, "yes, I am here">, <file2, "no, I am not here">>). In both cases, the string or strings are printed without surrounding quotes. In the first case the string is printed on the standard output; in the second case each of the strings is printed to the file specified by **filename,** which must be an fp atom. Using **nil** as a file name specifies the standard output. It is an error for the `main' function to return more than one instance of a given **filename.**

-p (parameters and options) specifies that if any command line options or parameters are present, the program should run immediately, with <> as input, instead of waiting for input from stdin. The arguments can then be opbtained by calling the procedure arguments (see later). If no command line arguments are given, input proceeds normally.

-a (AST system) can only be used with -m, and ignores -p. An AST system is an applicative state transition system. This option causes the procedure main to:

(1) apply the `main' function to <<>, <>>
(2) `main' must return <output, state>
(3) print the output part of the result
(4) read the standard input
(5) create a pair <data-read, state>
(6) call the `main' function on the pair
(7) `main' must return <output, new-state>
(8) print the output part of the result
(9) set state to the second part of the result
(10) if state is not nil, return to step (4)

Input and output are done as specified by any of the

-i and -o options. If -i is specified, each input will be a string of exactly one character, returned as soon as the character is available.

FILES

file.fp	input file
file.c	output file
fpc	compiler
libfp.a	run-time library
libdfp.a	run-time library to test implementation of the primitives
libnfp.a	run-time library for non-checking primitives

SEE ALSO

John Backus, **Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs** Turing Award lecture, Communications of the ACM, Volume 21, Number 8, August 1978

MODIFICATIONS

The major syntactic differences between the language accepted by FPC and the language described in the Backus paper are in the treatment of non-ascii characters. More specifically, the **define** character (three parallel strokes) disappears, the if character (arrow to the right) is entered as ->, the apply-to-all symbol (alpha) is entered as **aa**, the compose symbol is entered as **o** (lowercase O), and constants are preceded by an underscore instead of being overstruck. Also, bottom will be generated by a call to the primitive function **error**, characters are entered as 'c or '\c, and strings (vectors of characters) are entered as "string". In addition to the normal (left) insert functional form /, FPC provides tree insert \/ and right insert \. Tree insert is used where the order of application is unimportant and may be expected to be more efficient on some systems or in some applications. Unary negation is given by **neg** and - (minus) only accepts pairs of numbers.

Since there is a one-to-one mapping between fp functions and C procedures, characters not allowed in C procedure names are not permitted in fp function names.

As an example, the function ! (factorial) defined in the

paper would be written as

```
Def sub1 - o [id, _1]
Def eq0 eq o [id, _0]
Def fac eq0 -> _1; * o [id, fac o sub1]
```

NEW FUNCTIONS

FPC provides several functions that do not appear in the Backus paper, and several more are planned. The description of the currently available ones follows.

append merges any number of vectors into a single vector. Any top-level nils disappear. **trunc** is the floor function, it converts a real to the nearest integer that is less than or equal to it. **newline** is a constant-valued function which returns the string that signals a new line on the local system. The value returned is a string instead of a character since the system may require several characters (e.g., <CR, LF>) to signal an end of line. **implode** accepts an input string and returns a symbol the name of which is the same as the input string. **explode** is the corresponding function which accepts a symbol and returns the string corresponding to the symbol's name.

The function **arguments** returns the command line arguments, if any, in the order given; normal arguments are returned as strings, options are returned as the pair <option-char, string>, where string is the value of the option, if any, or nil otherwise.

trace is an output function: it is functionally identical to id, except that it only accepts strings as input. As a side effect, the string is printed on the standard output with no quotes around it. The program cannot redirect the output to a file.

The following functions all take as input a string representing a file name. **filetype** returns a symbol from the set none, empty, data, text, binary if the file does not exist, has no data, contains a valid FP object, contains text, or contains non-textual characters, respectively. All data files could be read as text files.

readfile returns the FP object read from the given file.

inputfile returns a string holding the text that was read from the given file.

DIAGNOSTICS

Unless -lnfp is used, programs check that the number of storage cells they allocated and returned was the same, and complain if that is not the case, i.e. if there was an error in the reference counting.

Whenever bottom is encountered, the stack is dumped to stderr (if -n and -lnfp were not used), together with the inputs to each of the functions on the stack. This can be a large amount of data.

The function **checkpoint** is functionally identical to the primitive id but outputs its argument to the output stream. This is helpful for tracing data flow in functions.

BUGS

Please report any bugs to the author.

AUTHOR

Ed Biagioni